



# Fundamentos de Python

**Lección 4: Distribución de paquetes.  
Integración con otros lenguajes.**

# ÍNDICE

1. Presentación y objetivos .....	2
2. Distribución de paquetes.....	3
Distribución de paquetes con setuptools.....	4
Caso práctico para crear un paquete .....	8
3. Integración de Python con otros lenguajes de programación .....	15
Integración de Python y R.....	15
4. Puntos clave.....	23

# Distribución de paquetes. Integración con otros lenguajes.

## 1. PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos a empaquetar nuestros programas implementados en el lenguaje de programación Python. La creación de estos paquetes es de utilidad para importarlos de una forma cómoda en futuras implementaciones que realicemos, así como distribuir nuestro código a la comunidad abierta de desarrolladores.

Por otro lado, estudiaremos cómo utilizar código procedente de otros lenguajes de programación como R en nuestras implementaciones de Python.



### Objetivos

- Conocer cómo empaquetar nuestras implementaciones, de modo que sea accesible por otros desarrolladores o para nuestras propias implementaciones.
- Conocer cómo instalar y utilizar un paquete creado por nosotros mismos.
- Conocer las herramientas actuales que nos permite integrar código de otros lenguajes de programación dentro de implementaciones realizadas en Python.

## 2. DISTRIBUCIÓN DE PAQUETES

Resulta de interés que, cada vez que implementemos un conjunto de programas software para resolver un determinado problema podamos exportarlos de forma que puedan ser distribuibles, tanto para facilitar nuestra tarea en un futuro, como para ofrecer a la comunidad de desarrolladores un nuevo módulo (implementado por nosotros mismos) que les sea de ayuda para resolver un determinado problema.



### ***Presta atención***

**Decidir distribuir nuestro código de forma pública puede ser una opción muy interesante. De este modo podremos facilitar las tareas de otros desarrolladores, colaborando en la creación y mejora de los repositorios de Python.**

Por ejemplo, imaginen el caso de la librería *matplotlib*, para realizar gráficos. Esta librería podemos instalarla por medio del gestor de paquetes *pip*, posteriormente la podemos importar y utilizar en nuestro código. Esta misma tarea es la que deseamos mostrar en esta lección: cómo empaquetar un proyecto realizado en Python, de modo que pueda ser instalado y utilizado por otro usuario.

Existen diferentes maneras para realizar esta tarea, como es el ejemplo del módulo *setuptools*.

## Distribución de paquetes con setuptools

Cada software que desarrollemos puede ser empaquetado para utilizarlo en otros proyectos propios o publicarlo en el *Python Package Index* (PyPI), de modo que nuestro software estará disponible para una gran comunidad de desarrolladores, ejecutando el comando `pip install <nombre_paquete>`.

Para poder empaquetar nuestro proyecto, éste debe tener la siguiente estructura:

DIRECTORIO\_DEL\_PROYECTO

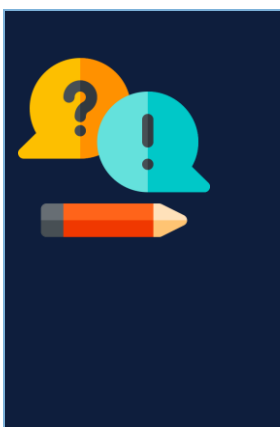
- LICENSE
- MANIFEST.in
- README.txt
- setup.py
- NOMBRE\_DEL\_PAQUETE
  - \_\_init\_\_.py
  - ARCHIVO1.py
  - ARCHIVO2.py
  - ARCHIVO\_N.py

Dónde cada uno de los elementos anteriores hacen referencia a:

- DIRECTORIO\_DEL\_PROYECTO: directorio donde se encuentra nuestro proyecto desarrollado en Python.
- NOMBRE\_DEL\_PAQUETE: este directorio debe tener el mismo nombre que el que le queramos poner al paquete que queremos empaquetar. Dentro de este directorio se encuentran todos los archivos que conforman nuestro proyecto. En este caso
- LICENSE: es el fichero donde se detallan los términos de licencia utilizados en el proyecto. Esta información es muy importante, ya que les indicará a terceros usuarios que utilicen nuestro software bajo qué condiciones pueden utilizarlo.
- MANIFEST.in: aquí se definen los criterios de inclusión y exclusión de archivos a su distribución de código fuente del proyecto.
- README.txt: aquí definiremos la documentación general de nuestro proyecto. Describiremos en qué consiste nuestro paquete y cómo utilizarlo.
- setup.py: este archivo hace referencia al script de instalación del paquete.

Para comenzar a empaquetar nuestro código, necesitamos tener instaladas dos librerías:

- **Wheel.** Herramienta utilizada para empaquetar el código.
  - Si no tuvieran disponible el módulo Wheel, ejecuten el siguiente comando: *pip install wheel*.
- **Twine.** Herramienta para subir los proyectos a PYPI.
  - En caso de no contar con esta herramienta, pueden instalarla como sigue: *pip install twine*.



### Conceptos

**PyPI es el repositorio de software oficial para aplicaciones de terceros en el lenguaje de programación Python. Los desarrolladores de Python pretenden que sea un catálogo exhaustivo de todos los paquetes de Python escritos en código abierto. (<https://pypi.org/>)**

Una vez mostrada la base teórica sobre cómo empaquetar nuestro código, procedemos a mostrar un ejemplo práctico, para mostrar todo el proceso paso a paso. En este caso práctico, crearemos un módulo llamado **Pyoperaciones**. El código se encuentra en un único fichero .py y contiene un conjunto de funciones para realizar operaciones básicas con números enteros, y que podremos utilizar en cualquier programa que importemos dicho módulo.

Como comentamos al inicio de esta lección, el primer paso que debemos realizar consiste en crear un conjunto de directorios y ficheros de modo que nuestro proyecto quede bien organizado.

En este caso práctico, hemos creado un directorio bajo el nombre **pyoperaciones** con el siguiente contenido:

- setup.py. Script de instalación. (Ver ejemplo en imagen 1)
- README.md. Archivo de texto en formato Markdown con una descripción de nuestro proyecto.
- MANIFEST.in. Archivo con la lista de archivos que se incluyen/excluyen de nuestro proyecto.
- LICENSE.txt. Archivo de texto con la licencia que hemos elegido para el proyecto.
- pyoperaciones (directorio del paquete). Contiene las fuentes .py del proyecto. Este es probablemente el directorio de mayor importancia, ya que contiene el paquete a distribuir. En nuestro caso, contiene un archivo `__init__.py` con todo el código fuente de Pyoperaciones. Es esencial que el código que implementemos esté documentado correctamente para que cualquier usuario que quiera utilizar nuestro paquete pueda consultar la ayuda.

```

"""Instalador para el paquete "pyoperaciones"."""

from setuptools import setup

long_description = (
    open('README.txt').read()
    + '\n' +
    open('LICENSE').read()
    + '\n')

setup(
    name="pyoperaciones",
    version="0.1",
    description="A tool to perform mathematical operations.",
    long_description=long_description,
    classifiers=[
        # Indica la estabilidad del proyecto. Los valores comunes son
        # 3 - Alpha
        # 4 - Beta
        # 5 - Production/Stable
        "Development Status :: 4 - Beta",
        # Indique a quien va dirigido su proyecto
        "Intended Audience :: Developers",
        "Topic :: Utilities",
        # Indique licencia usada (debe coincidir con el "license")
        "License :: OSI Approved :: GNU General Public License v3 (GPLv3)",
        # Indique versiones soportadas, Python 2, Python 3 o ambos.
        "Programming Language :: Python :: 3",
        "Operating System :: OS Independent",
    ],
    keywords="pyoperaciones mathematical operations",
    author="Ramón Rueda",
    author_email="ramonruedadelgado@gmail.com",
    license="GNU GPLv3",
    packages=["pyoperaciones"]
)

```

Figura 2.1: Ejemplo setup.py

## Caso práctico para crear un paquete

Existen diversas formas para crear un paquete. En nuestro caso particular utilizaremos la herramienta **wheel**, debido a su sencillez y rapidez. Para construir un paquete con dicha herramienta, debemos ejecutar el siguiente comando en una terminal:

```
python ./setup.py bdist_wheel
```

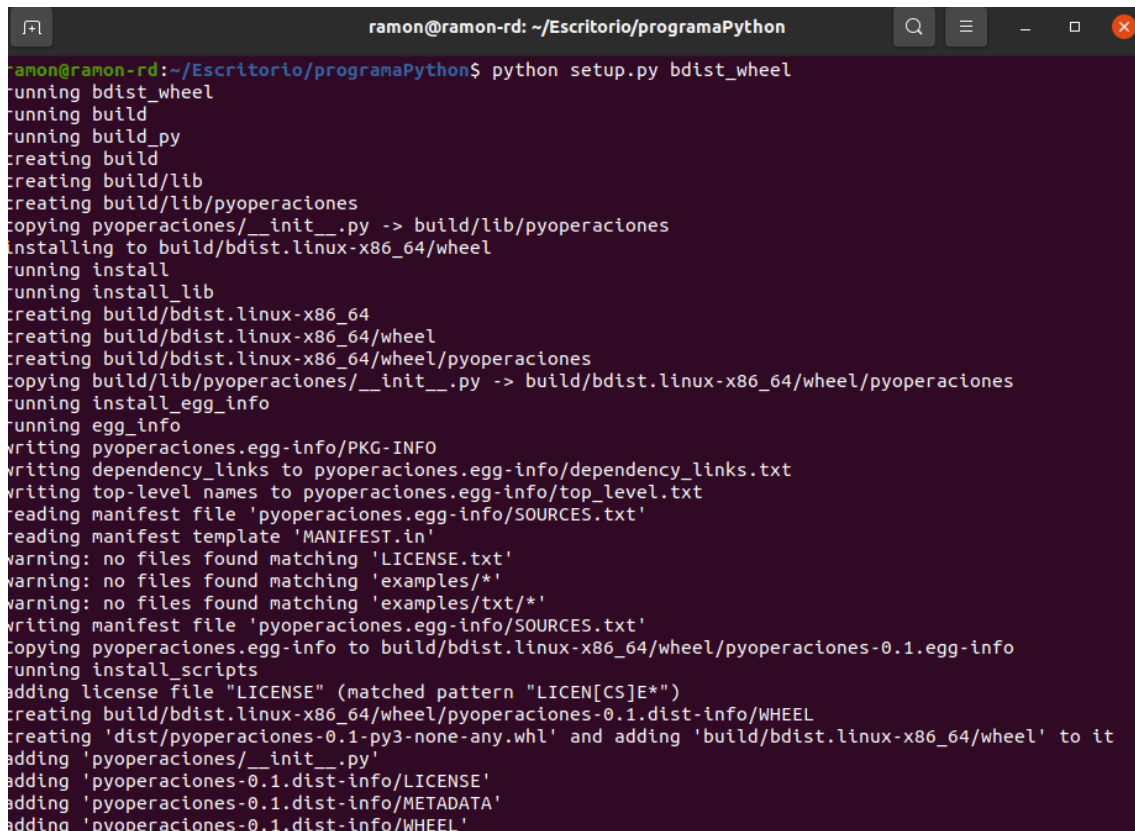


### *Importante*

Existen diferentes mecanismos para empaquetar nuestras implementaciones en Python y crear una librería. Aunque en esta lección se muestra un ejemplo práctico usando **wheel**, se anima al lector a explorar otros mecanismos, como pueden ser las herramientas **egg**, **rpm**, etc.

En relación al comando anterior, **bdist\_wheel** construirá un paquete con la versión adecuada de Python (2.x o 3.x). Una vez finalice el proceso, se habrá creado un nuevo directorio llamado "dist" que incluirá el paquete creado por **wheel** "nombre\_paquete-version-py3-none-any.whl". Además, se creará un directorio adicional "build" con los archivos generados durante el proceso de generación.





```

ramon@ramon-rd: ~/Escritorio/programaPython
ramon@ramon-rd:~/Escritorio/programaPython$ python setup.py bdist_wheel
running bdist_wheel
running build
running build_py
creating build
creating build/lib
creating build/lib/pyoperaciones
copying pyoperaciones/__init__.py -> build/lib/pyoperaciones
installing to build/bdist.linux-x86_64/wheel
running install
running install_lib
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/wheel
creating build/bdist.linux-x86_64/wheel/pyoperaciones
copying build/lib/pyoperaciones/__init__.py -> build/bdist.linux-x86_64/wheel/pyoperaciones
running install_egg_info
running egg_info
writing pyoperaciones.egg-info/PKG-INFO
writing dependency_links to pyoperaciones.egg-info/dependency_links.txt
writing top-level names to pyoperaciones.egg-info/top_level.txt
reading manifest file 'pyoperaciones.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no files found matching 'LICENSE.txt'
warning: no files found matching 'examples/*'
warning: no files found matching 'examples/txt/*'
writing manifest file 'pyoperaciones.egg-info/SOURCES.txt'
copying pyoperaciones.egg-info to build/bdist.linux-x86_64/wheel/pyoperaciones-0.1.egg-info
running install_scripts
adding license file "LICENSE" (matched pattern "LICEN[CS]E*")
creating build/bdist.linux-x86_64/wheel/pyoperaciones-0.1.dist-info/WHEEL
creating 'dist/pyoperaciones-0.1-py3-none-any.whl' and adding 'build/bdist.linux-x86_64/wheel' to it
adding 'pyoperaciones/__init__.py'
adding 'pyoperaciones-0.1.dist-info/LICENSE'
adding 'pyoperaciones-0.1.dist-info/METADATA'
adding 'pyoperaciones-0.1.dist-info/WHEEL'

```

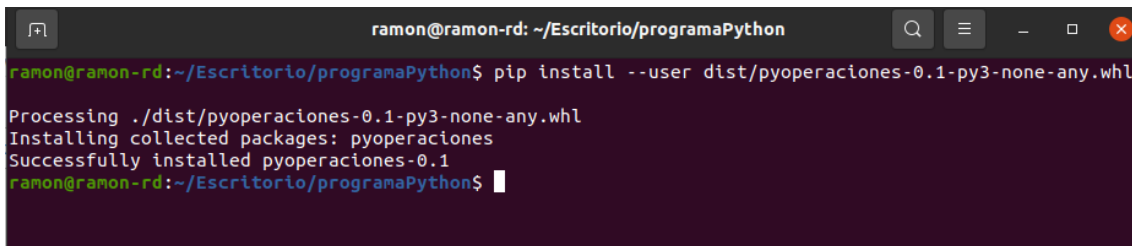
Figura 2.2: Ejemplo creación de un paquete usando wheel

Una vez que hemos creado nuestro paquete, tenemos dos opciones:

- Instalarlo en nuestra máquina para comprobar su correcto funcionamiento.
- Subir el paquete del proyecto a PyPI.

Para el primer caso, ejecutaremos la siguiente orden en una terminal:

```
pip install --user ./dist/pyoperaciones-0.1-py3-none-any.whl
```

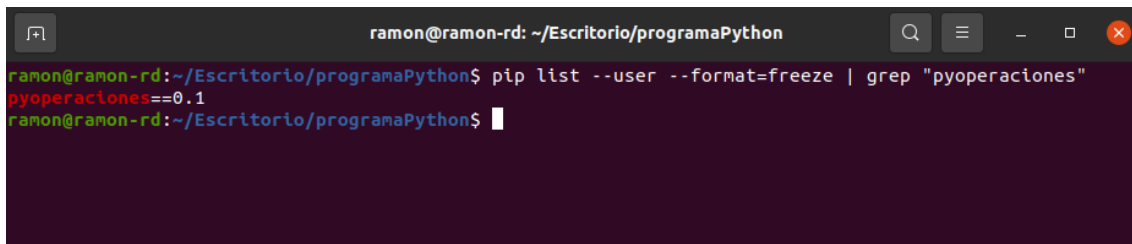
A terminal window titled 'ramon@ramon-rd: ~/Escritorio/programaPython' with search, menu, and window control icons. The command 'pip install --user dist/pyoperaciones-0.1-py3-none-any.whl' is entered. The output shows the package being processed and successfully installed.

```
ramon@ramon-rd:~/Escritorio/programaPython$ pip install --user dist/pyoperaciones-0.1-py3-none-any.whl
Processing ./dist/pyoperaciones-0.1-py3-none-any.whl
Installing collected packages: pyoperaciones
Successfully installed pyoperaciones-0.1
ramon@ramon-rd:~/Escritorio/programaPython$
```

*Figura 2.2: Instalación del paquete en desarrollo*

Para comprobar si el paquete ha sido instalado correctamente ejecutamos el siguiente comando:

```
pip list --user --format=freeze | grep "pyoperaciones"
```

A terminal window titled 'ramon@ramon-rd: ~/Escritorio/programaPython' with search, menu, and window control icons. The command 'pip list --user --format=freeze | grep "pyoperaciones"' is entered. The output shows 'pyoperaciones==0.1' in red text.

```
ramon@ramon-rd:~/Escritorio/programaPython$ pip list --user --format=freeze | grep "pyoperaciones"
pyoperaciones==0.1
ramon@ramon-rd:~/Escritorio/programaPython$
```

*Figura 2.3: Comprobación de la correcta instalación de nuestro paquete*

A continuación se muestra un ejemplo de importación y uso del paquete creado:

```
1 import pyoperaciones as po
2 |
3 print("La suma de 2 y 3 es: ", po.sumaEnteros(2,3))
4
5 print("La división de 8 entre 4 es: ", po.divideEnteros(8, 4))
```

Figura 2.4: Programa de prueba. Se importa el módulo `pyoperaciones` y se muestra un ejemplo de uso

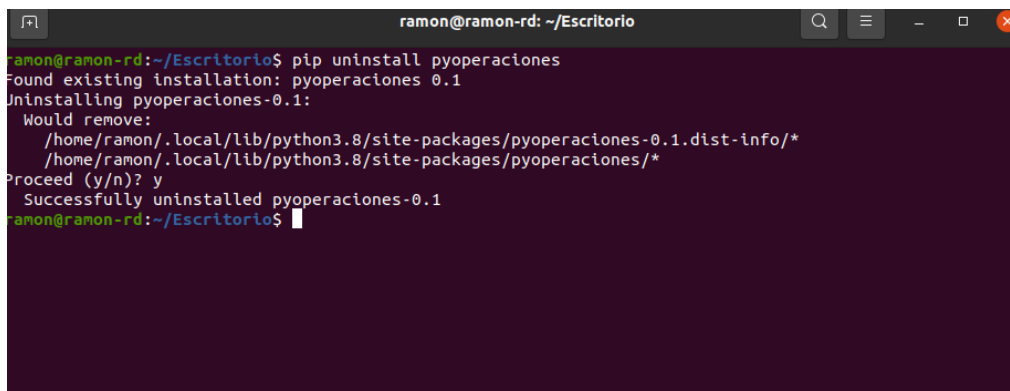
A screenshot of a terminal window with a dark background. The window title is "ramon@ramon-rd: ~/Escritorio". The prompt is "ramon@ramon-rd:~/Escritorio\$". The command "python pruebaOperaciones.py" has been entered. The output shows two lines: "La suma de 2 y 3 es: 5" and "La división de 8 entre 4 es: 2.0". The prompt is now "ramon@ramon-rd:~/Escritorio\$" with a cursor.

```
ramon@ramon-rd:~/Escritorio$ python pruebaOperaciones.py
La suma de 2 y 3 es: 5
La división de 8 entre 4 es: 2.0
ramon@ramon-rd:~/Escritorio$
```

Figura 2.5: Ejecución de programa de prueba y visualización de resultados

Si quisiéramos eliminar el paquete que acabamos de instalar, ejecutamos la siguiente orden:

*`pip uninstall pyoperaciones`*



```

ramon@ramon-rd: ~/Escritorio
ramon@ramon-rd:~/Escritorio$ pip uninstall pyoperaciones
Found existing installation: pyoperaciones 0.1
Uninstalling pyoperaciones-0.1:
  Would remove:
    /home/ramon/.local/lib/python3.8/site-packages/pyoperaciones-0.1.dist-info/*
    /home/ramon/.local/lib/python3.8/site-packages/pyoperaciones/*
Proceed (y/n)? y
Successfully uninstalled pyoperaciones-0.1
ramon@ramon-rd:~/Escritorio$

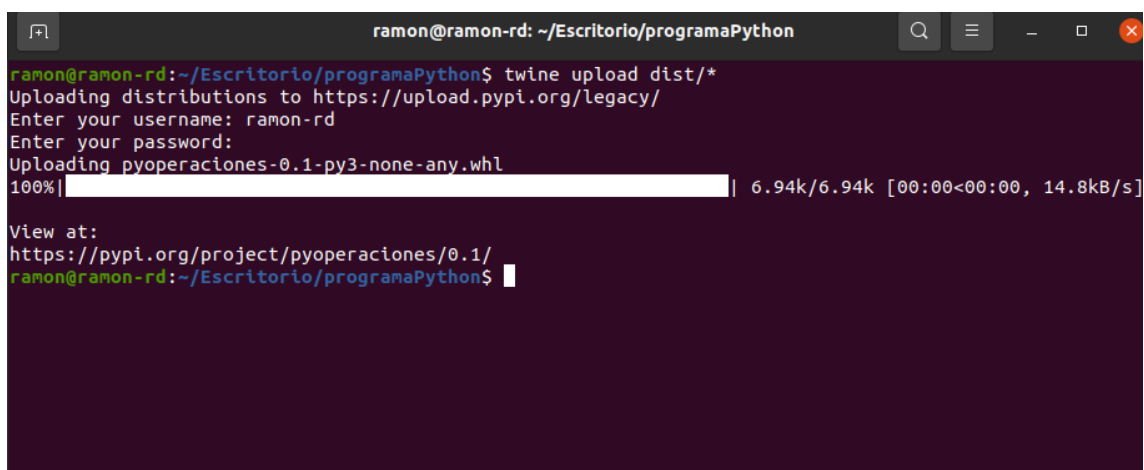
```

Figura 2.6: Eliminar un paquete instalado

Para subir el proyecto a PyPI debemos seguir los siguientes pasos:

1. En primer lugar, debemos completar el formulario de registro de Pypi (<https://pypi.org/account/register/>)
2. Subir el paquete a PyPI. Para ello, ejecutaremos en una terminal lo siguiente:

*twine upload dist/\**



```

ramon@ramon-rd: ~/Escritorio/programaPython
ramon@ramon-rd:~/Escritorio/programaPython$ twine upload dist/*
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username: ramon-rd
Enter your password:
Uploading pyoperaciones-0.1-py3-none-any.whl
100%|████████████████████████████████████████| 6.94k/6.94k [00:00<00:00, 14.8kB/s]

View at:
https://pypi.org/project/pyoperaciones/0.1/
ramon@ramon-rd:~/Escritorio/programaPython$

```

Figura 2.7: Subida del proyecto a PyPI

Ahora podremos instalar nuestro paquete desde cualquier máquina:

```
ramon@ramon-rd:~/Escritorio/programaPython$ pip install pyoperaciones
Collecting pyoperaciones
  Downloading pyoperaciones-0.1-py3-none-any.whl (2.7 kB)
Installing collected packages: pyoperaciones
Successfully installed pyoperaciones-0.1
ramon@ramon-rd:~/Escritorio/programaPython$
```

*Figura 2.8: Instalación del paquete*

### 3. INTEGRACIÓN DE PYTHON CON OTROS LENGUAJES DE PROGRAMACIÓN

Python ofrece la posibilidad de utilizar códigos implementados en otros lenguajes de programación dentro de un programa implementado en Python.

Podemos encontrar, por ejemplo, la librería rpy2 que nos permite integrar código realizado en R en nuestras implementaciones de Python.

#### Integración de Python y R

Tanto R como Python son dos lenguajes de programación que cuentan con una serie de herramientas que pueden ser de ayuda para resolver tareas relacionadas con ciencia de datos. Como hemos estudiado en lecciones previas, elegir el lenguaje de programación que vamos a utilizar para resolver un determinado problema es un paso clave, ya que si seleccionamos un lenguaje de programación que cuente con una gran variedad de librerías y módulos, este podrá ayudarnos a resolver nuestras tareas de una forma más sencilla. Sin embargo, la realidad es que a veces necesitamos utilizar más de un lenguaje de programación, ya que uno nos ofrece unas ventajas que otro lenguaje no puede ofrecernos.



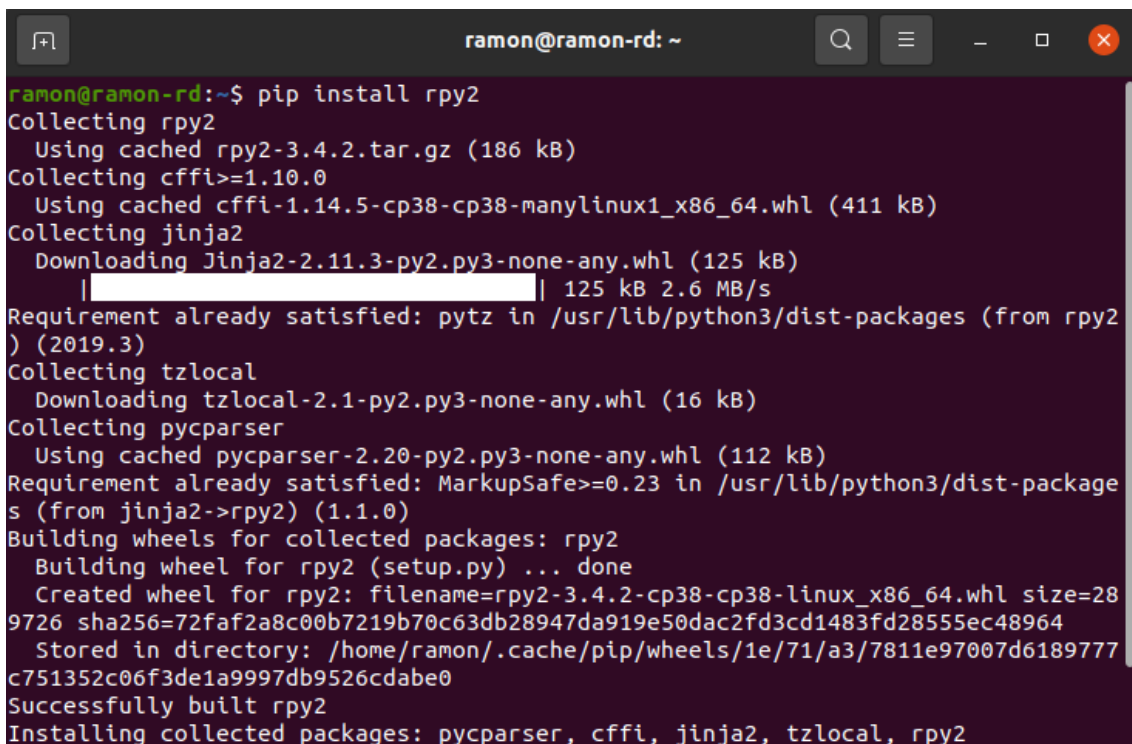
#### *Importante*

**La integración de código nos permite explotar al máximo las ventajas de varios lenguajes de programación y unificarlos en un único programa.**

**Esta integración nos permitirá ahorrar tiempo de desarrollo, ya que podremos hacer uso de paquetes que hayan sido implementados en otro lenguaje de programación.**

Es el caso de los lenguajes de programación Python y R. Mientras que Python dispone de una gran variedad de herramientas para el cálculo numérico y aprendizaje automático (entre otros), R dispone de un gran ecosistema para el análisis estadístico. Por este motivo, poder trabajar de forma conjunta con dos lenguajes de programación al mismo tiempo, puede ofrecernos grandes oportunidades, ya que aprovecharemos las ventajas de cada lenguaje de programación, minimizando de este modo los inconvenientes de cada uno de ellos.

Para poder trabajar con R y Python de forma simultánea, debemos instalar el paquete **rpy2** como sigue:

A terminal window titled 'ramon@ramon-rd: ~' showing the command 'pip install rpy2' and its output. The output lists the collection of rpy2, cffi, and Jinja2, the downloading of Jinja2, the satisfaction of requirements for pytz and MarkupSafe, the collection of tzlocal and pycparser, the building of wheels for rpy2, and the successful installation of rpy2 and its dependencies.

```
ramon@ramon-rd:~$ pip install rpy2
Collecting rpy2
  Using cached rpy2-3.4.2.tar.gz (186 kB)
Collecting cffi>=1.10.0
  Using cached cffi-1.14.5-cp38-cp38-manylinux1_x86_64.whl (411 kB)
Collecting Jinja2
  Downloading Jinja2-2.11.3-py2.py3-none-any.whl (125 kB)
    | 125 kB 2.6 MB/s
Requirement already satisfied: pytz in /usr/lib/python3/dist-packages (from rpy2) (2019.3)
Collecting tzlocal
  Downloading tzlocal-2.1-py2.py3-none-any.whl (16 kB)
Collecting pycparser
  Using cached pycparser-2.20-py2.py3-none-any.whl (112 kB)
Requirement already satisfied: MarkupSafe>=0.23 in /usr/lib/python3/dist-packages (from Jinja2->rpy2) (1.1.0)
Building wheels for collected packages: rpy2
  Building wheel for rpy2 (setup.py) ... done
  Created wheel for rpy2: filename=rpy2-3.4.2-cp38-cp38-linux_x86_64.whl size=289726 sha256=72faf2a8c00b7219b70c63db28947da919e50dac2fd3cd1483fd28555ec48964
  Stored in directory: /home/ramon/.cache/pip/wheels/1e/71/a3/7811e97007d6189777c751352c06f3de1a9997db9526cdabe0
Successfully built rpy2
Installing collected packages: pycparser, cffi, Jinja2, tzlocal, rpy2
```

Figura 3.9: Instalación del paquete rpy2



### *Importante*

Para utilizar **rpy2** necesitarás tener instalado tanto Python como R, además de las librerías R que quieras utilizar.

Una vez instalado, podemos crear un programa e importar la librería **rpy2**, que nos permitirá emplear utilizar código R dentro del código Python. Pero antes de nada, ¿qué es rpy2? **Rpy2** es una interfaz que permite que podamos comunicar información entre R y Python y que podamos acceder a las funcionalidades de R desde Python. Por lo tanto, podemos estar empleando Python para el desarrollo de todo nuestro programa, y en caso de que necesitemos emplear cualquier librería estadística de R, podremos acceder a misma usando **rpy2**.

En concreto, debemos importar el objeto `r` de `rpy2.robjects`. Este objeto recibirá como cadena de texto el conjunto de comandos en R que queramos ejecutar, como por ejemplo imprimir el mensaje "Hello World":

```
from rpy2.robjects import r
r('print("Hello world!")')
```

*Figura 3.2: Ejemplo sencillo de integración de código R en Python*



A terminal window titled 'ramon@ramon-rd: ~/Escritorio'. The prompt is 'ramon@ramon-rd:~/Escritorio\$'. The command 'python integracionPython\_R.py' has been entered. The output is '[1] "Hello world!"'. The prompt is now 'ramon@ramon-rd:~/Escritorio\$' with a cursor.

```
ramon@ramon-rd: ~/Escritorio
ramon@ramon-rd:~/Escritorio$ python integracionPython_R.py
[1] "Hello world!"
ramon@ramon-rd:~/Escritorio$
```

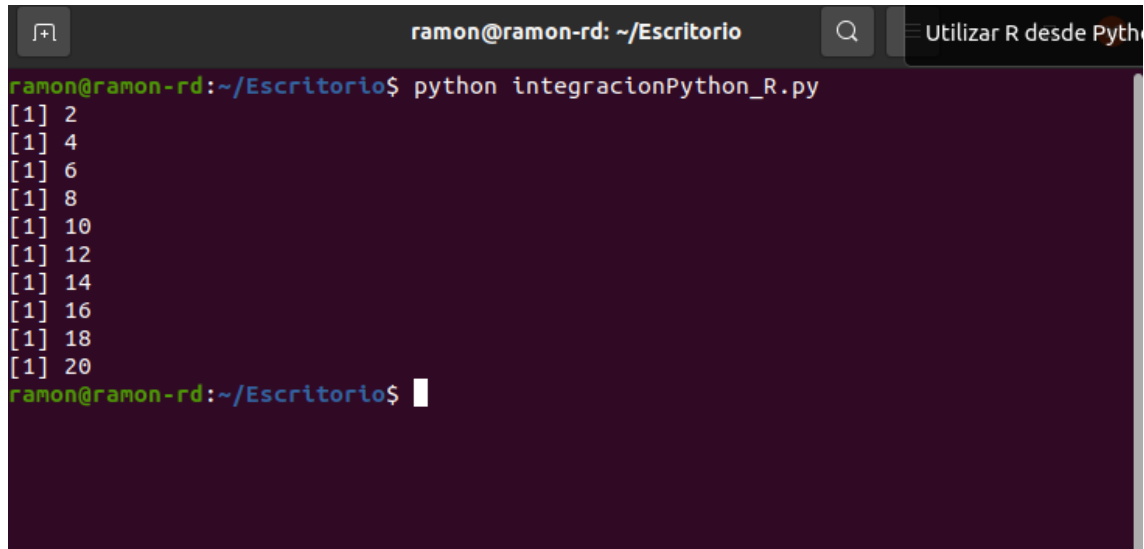
Figura 3.3: Ejecución ejemplo sencillo (hola mundo)

Un ejemplo un poco más complicado: mostrar los números pares del 1 al 20, utilizando código R incrustado en Python:

```
from rpy2.objects import r

r('''
n <- 20
for (i in 1:n){
  if(i%%2 == 0){
    print(i)
  }
}
''')
```

Figura 3.4: Ejemplo números pares



```
ramon@ramon-rd: ~/Escritorio
ramon@ramon-rd:~/Escritorio$ python integracionPython_R.py
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
[1] 12
[1] 14
[1] 16
[1] 18
[1] 20
ramon@ramon-rd:~/Escritorio$
```

Figura 3.5: Ejecución programa números pares

Sin embargo, para poder sacar todo el potencial de esta herramienta, sería interesante poder intercambiar datos entre el código implementado en Python y en R. De este modo, los resultados obtenidos en R podrían ser utilizados como entrada para un programa en Python, o viceversa. Para asignar a una variable en R un valor de Python, lo haremos por medio del método **assign**. Este método recibe dos argumentos, el primero hace referencia al nombre de la variable en R y el segundo al valor.

Si quisiéramos recuperar los valores de R en Python, bastaría con emplear la función del objeto **r**. Por ejemplo, el siguiente fragmento de código asigna desde Python un valor a una variable en R, se realiza una serie de operaciones matemáticas, y por último se devuelve el valor calculado a una variable de Python.

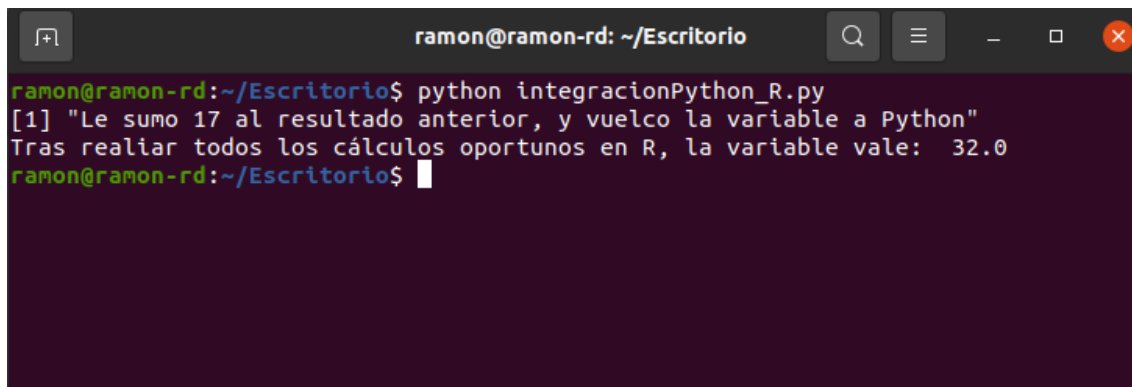
```
from rpy2.robjects import r

a = 5

r.assign('v', a)
r('z <- v*3')
r('sprintf("La variable vale ahora %i", z)')
r('z <- z+17')
r('print("Le sumo 17 al resultado anterior, y vuelco la variable a Python")')

b = r('z')
print("Tras realizar todos los cálculos oportunos en R, la variable vale: " , b[0])
```

Figura 3.6: Ejemplo de asignación de variables entre Python y R



```
ramon@ramon-rd: ~/Escritorio
ramon@ramon-rd:~/Escritorio$ python integracionPython_R.py
[1] "Le sumo 17 al resultado anterior, y vuelco la variable a Python"
Tras realizar todos los cálculos oportunos en R, la variable vale:  32.0
ramon@ramon-rd:~/Escritorio$
```

Figura 3.7: Ejecución del ejemplo de integración de variables entre Python y R

Cabe destacar que el código R que utilizemos en nuestras implementaciones de Python puede ser almacenado en una variable, de modo que podamos utilizar fragmentos de código R a lo largo de nuestro programa Python de una forma más cómoda. Para el ejemplo de obtener los 20 primeros números pares, vamos a crear una función en R que reciba como parámetro cuántos números pares queremos calcular, y lo almacenaremos en una variable de Python.

```
import rpy2.robjects as ro

codigo_r = """
pares <- function(n){
  for (i in 1:n){
    if(i%%2 == 0){
      print(i)
    }
  }
}
"""
ro.r(codigo_r)

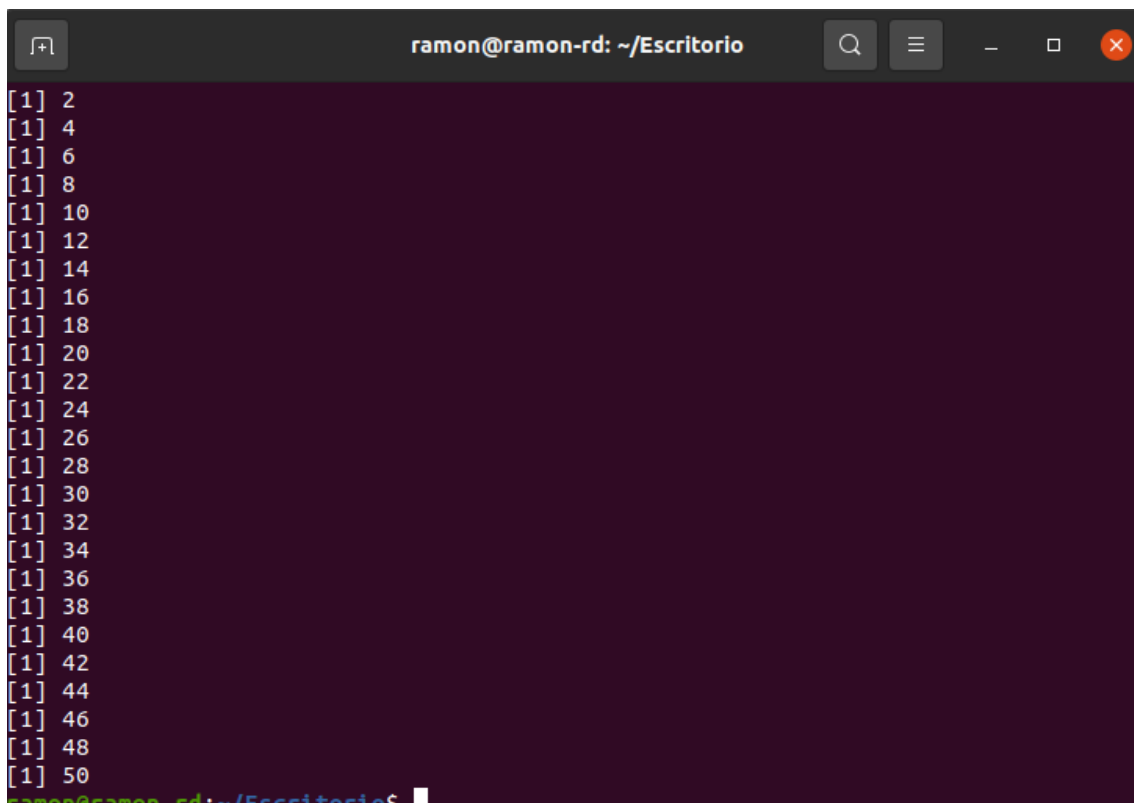
pares_py = ro.globalenv['pares']

pares_py(50)
```

Figura 3.8: Almacenar código R en una variable de Python

En el código anterior hemos definido en R la función *pares*. Esta función recibe como entrada un entero *n*, que indica hasta qué número queremos calcular los números pares. Posteriormente, hacemos uso de la función *globalenv* de la librería *rpy2* para poder extraer la función *pares* definida en R, y almacenarla en nuestra implementación de Python; en concreto en la variable de *pares\_py*.

En este punto, la variable *pares\_py* hace referencia a la función implementada en R, por lo que haciendo una llamada a *pares\_py*, estaremos invocando directamente a la función que implementamos en R. A continuación se muestra un ejemplo de la salida obtenida tras calcular los números pares que hay comprendidos entre 1 y 50.



```
ramon@ramon-rd: ~/Escritorio
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
[1] 12
[1] 14
[1] 16
[1] 18
[1] 20
[1] 22
[1] 24
[1] 26
[1] 28
[1] 30
[1] 32
[1] 34
[1] 36
[1] 38
[1] 40
[1] 42
[1] 44
[1] 46
[1] 48
[1] 50
ramon@ramon-rd: ~/Escritorio
```

Figura 3.9: Cálculo de los números pares comprendidos entre 1 y 50.



### ***Presta atención***

Existe una gran variedad de funciones y operaciones que pueden realizarse usando la librería rpy2. Se recomienda al usuario consultar la documentación de dicho paquete para ampliar más información que le resulte de interés:

<https://rpy2.github.io/doc/latest/html/index.html>

## 4. PUNTOS CLAVE

En esta lección hemos aprendido:

- Cuáles son las ventajas de distribuir nuestro código.
- El proceso a seguir para empaquetar nuestro código y crear un paquete distribuible. Este paquete puede ser utilizado exclusivamente por nosotros, o podemos incluirlo en el repositorio público PyPI para que cualquier usuario pueda beneficiarse de nuestros desarrollos.
- En qué consiste y cuáles son las ventajas de integrar códigos de programación implementados en otro lenguaje en nuestros desarrollos de Python.
- Cómo integrar código desarrollado en R en programas implementados en Python.

